CROSS-LAYER SCHEDULING IN CLOUD COMPUTING SYSTEMS

BY

HILFI MADARI ALKAFF

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Advisor:

Professor Indranil Gupta

Abstract

Today, application schedulers are decoupled from routing level schedulers, leading to sub-optimal throughput for cloud computing platforms. In this thesis, we propose a cross-layer scheduling framework that bridges the application level scheduler with the routing level scheduler (SDN). We realize our framework in a batch-processing framework (Hadoop [1]) and a stream-processing framework (Storm [2]). Our experimental results show that we are able to improve throughput of jobs in Storm and Hadoop by up to 32% and 29% respectively.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Overview

Recent years have witnessed the emergence of applications and services that generate massive collections of data (also known as Big Data) such as social media and e-commerce. To analyze data quickly and efficiently and extract value for customers, these services use distributed frameworks such as Map-Reduce and Storm which are being deployed in cloud environments. The frameworks split the data across clusters of hundreds or thousands of computers and perform operations potentially split into multiple stages. To reduce the running costs of the cloud provider (who manages the infrastructure) and the customer (who pays by the hour), it is important to improve cluster utilization and keep the completion time of distributed computing jobs low.

However, there are multiple challenges that need to be overcome to run generic distributed computing applications efficiently. Firstly, currently applications are not aware of the underlying networking topology. Their scheduling algorithms currently only take into account the data-locality, CPU consumption and memory consumption of machines but they do not take into account the amount of bandwidth that is available in the datacenter. Thus, if new network links, fail, are brought down or introduced between physical routers, the applications will not be aware of it and might schedule worker tasks into machines that have lower outgoing network bandwidth. To be able to support responsive scheduling decisions, we also need to ensure that updates made to and receive from the routers are fast so that we could adapt our cross-layer scheduling policies accordingly.

The second challenge is that there are a variety of applications, ranging from batch to stream processing. Although these frameworks are solving

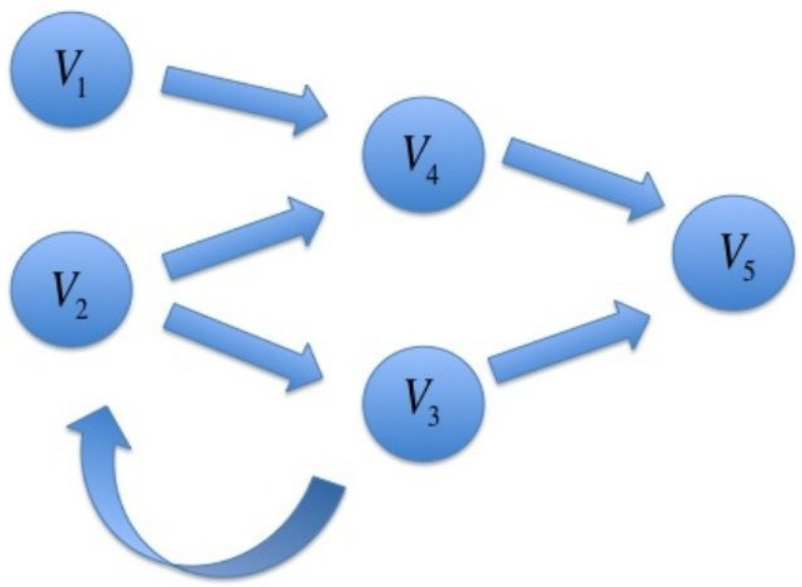

Figure 1.1: Typical Data Flow for Stream Processing Applications. Each of the vertices represent a computation node.

different problems, they are essentially dataflow programs. Batch processing frameworks typically use Map-Reduce paradigm [3] while stream processing frameworks in which running applications are decomposed into multiple vertices in a directed-graph as shown in Figure 1.1. The arrows in the graph indicate the direction of data path (i.e. $\mathcal{V}_1$ is sending data to $\mathcal{V}_4$). There is no restriction for the applications to construct this graph. As shown in the figure, even cycles are allowed. Each vertex in this graph represents a Computation-Node that takes input data from the incoming edge and produces output data on the outgoing edge while an edge from vertex $\mathcal{V}_i$ to vertex $\mathcal{V}_j$ in the graph indicates that the output of A will be sent to B as its input data. Typical operations in the graph include, but are not limited to, a filter, map, and join operation. Thirdly, users demand jobs to be more interactive while current datacenters span thousands of machines. Under such scalability constraints, exhaustive enumeration of all the solutions to compute the most optimal scheduling decision is not feasible. This calls for a radically different approach.

In this thesis, we design a cross-layer scheduling framework that tackles all of the above challenges to maximize throughput of data-processing framework applications. Our proposed framework operate in two-levels that complement

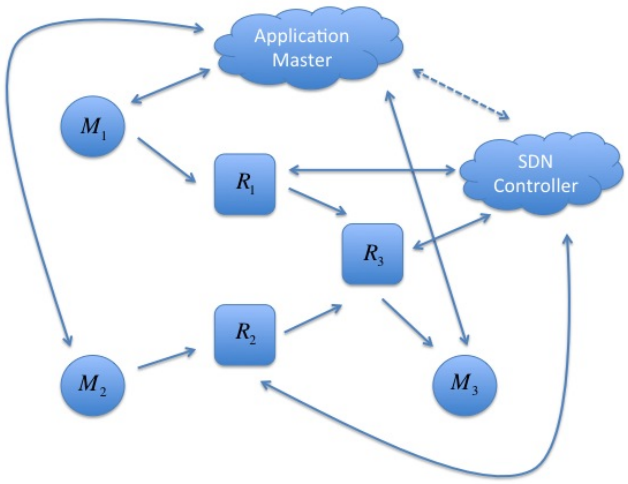

Figure 1.2: Application master is responsible for scheduling and monitoring worker machines, M_i, while SDN controller for the routers, R_i. The application master will consult the SDN controller to find out about the bandwidth available for the machines.

each other – application-level and routing-level scheduler. The application-level scheduler decides which physical machines a worker task should be allocated to. The lower level scheduler lies at the routing level in which the physical routers and links are interconnected. With regards to the multiple data-flow representations, we observe that data-paths in the Map-Reduce paradigm is a strict subset of data-paths in streaming framework since in the former, the graph can be divided into stages of maps and reduces as shown in Figure 1.3 where each of the mappers could send data to all of the reducers and vice versa while in the latter, there is no restriction in how to construct the graph. Thus, the Map-Reduce paradigm could be considered as a special case of the streaming framework. This tackles the first challenge.

Software-Defined Networking (SDN) [4] is a standard which makes network routing easier and more flexible by leveraging a centralized server, SDN Controller, that installs forwarding rules inside routers and monitors their status quickly. As measured in [5], it takes less than 5ms to install a single flow from a controller in an OpenFlow switch which is small compared to a traditional network.

At its core, our framework utilizes the Simulated Annealing [6] algorithm

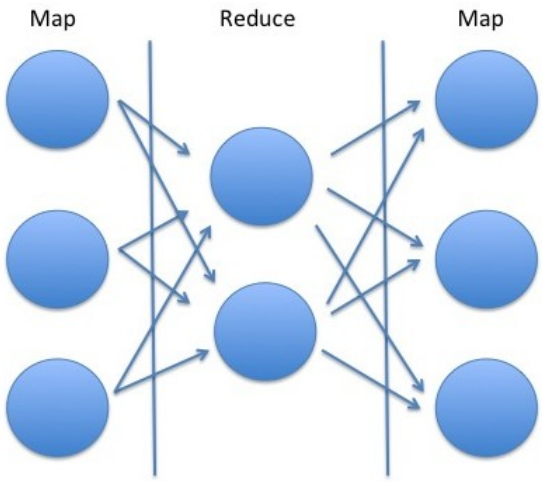

Figure 1.3: Data flow under Map-Reduce paradigm can be divided into stages of maps and reduces and on each stage.

for approximating the global optimization problem on both levels – application and routing. We choose this algorithm due to its simplicity of implementation, its property of escaping local optima, having a relatively short running time while still achieving good results. This tackles the third challenge of scalability.

Figure 1.2 shows an overview of our proposed framework. Current distributed computing platforms monitor the machines for executing user jobs while SDN controllers monitor the routers in the network topology as shown by the solid lines. In this thesis, we add the communication path between the application master and the SDN controller as shown by the dashed line. This enables the application make a better scheduling decision as it will now be aware of changes that happen in the routing level (e.g., when a router or a link fails how much bandwidth is available between any of its worker machines).

Data center architectures traditionally follow a three-tier architecture, e.g., FatTree [7], VL2 [8], DCell [9]. However, more recently, random topologies such as Scafida [10] and Jellyfish [11] have been proposed. To this end, we evaluate the framework that we design on both a hierarchical and non-hierarchical topology.

1.2 Technical Contributions

Our technical contributions in this thesis are as follows.

1. We design a novel cross-layer scheduling framework so that applications can make better scheduling decisions in a topology-aware manner.

2. We show that we can use a simple but effective algorithm, Simulated Annealing, to make fast and good cross-layer scheduling decisions.

3. We show that after implementing our cross-layer scheduling framework in two of the most widely used data processing framework, Hadoop and Storm, we are able to improve the throughput of the running tasks by up to 29% and 32% respectively.

1.3 Thesis Outline

This thesis is structured as follows. We begin with a background on the streaming frameworks dataflow graphs model followed by the design of our throughput optimization algorithm (Chapter 2). We then show our how do we implement it in Storm (Chapter 3) and our experimental results (Chapter 4). We will then survey related works in the area. (Chapter 5). Finally, we discuss some extensions that could further be made with our algorithm and conclude (Chapter 6).

CHAPTER 2

DESIGN

In this chapter, we will talk about the algorithm used in our cross-layer optimization framework and how will our framework reacts to failures in either link-level or machine-level.

2.1 Algorithm

The key idea in our algorithm is to separate the scheduling logic into a placement-level and a routing-level which complement each other. Then, in each of the level we execute the relevant algorithm and heuristic that is specific to that level. Consider the network topology that we operate on to be a graph $\mathcal{G}$ with the vertex $\mathcal{V}$ indicates the physical machines and the edge $\mathcal{E}$ indicates links between the machines. Each job $\mathcal{J}_i$ that will be running on this topology will have $\mathcal{T}_i$ tasks each.

When our data processing framework first starts, we run a modified Floyd-Warshall Algorithm [12] to compute all-pairs k-shortest paths between the hosts in the topology that the framework will be running on. The result of this computation is then cached in a hash-table indexed by the two machines that we are interested to finding the path for, which we call Topology-Map. This Topology-Map will then be consulted for future scheduling decision.

In a cluster of 1000 nodes, we calculated that recent network topology such as Jellyfish and Fat-Tree have at most 10 path hops between any two nodes (Paths without cycles). If we set the k parameter for our k-shortest path algorithm to 10, we find that the amount of memory used for this step is at most 1000^2 nodes $\times$ 10 hops/node $\times$ 10 different paths / 2 (Paths from A to B yield the same result as B to A) = 50MB. Typical RAM on today's machines are in the order of 8GB. As such, our storage requirement

Algorithm 1 Simulated Annealing Algorithm

```
1: function MAIN
2:     currentTemperature ← 100
3:     currentState ← initState()
4:
5:     for i ← 1 to maxStep do
6:         newState ← genState(currentState)
7:         newUtil ← computeUtil(newState)
8:
9:         if doTransition(currentUtil, newUtil) then
10:            currentState ← newState
11:            currentUtil ← newUtil
12:        end if
13:
14:        if currentUtil ≥ bestUtil then
15:            bestUtil ← currentUtil
16:            bestState ← currentState
17:        end if
18:        currentTemperature ← currentTemperature × 0.9
19:    end for
20:
21:    return bestState
22: end function
23:
24: function doTransition(oldUtil, newUtil, currentTemperature)
25:    if newUtil > oldUtil then
26:        return 1
27:    else
28:        return exp((newUtil − oldUtil)/currentTemperature)
29:    end if
30: end function
31:
32: function computeUtil(graph)
33:    sinkVertices ← Find the sink Computation-Node from Graph
34:    util ← 0
35:
36:    for i ← 1 to length(sinkVertices) do
37:        sinkVertexUtil ← Compute path bandwidth leading to
    sinkVertices(i)
38:        util ← util + sinkVertexUtil
39:    end for
40:
41:    return util
42: end function
```

Algorithm 2 Simulated Annealing Functions in the Placement Level

```
1: function initState(graph, machines)
2:     for i ← 1 to length(Graph.Computation-Node ()) do
3:         machine ← Pick a random machine in datacenter from machines
4:         Assign machine to run task i
5:     end for
6: end function
7:
8: function genState(graph, machines)
9:     chosenVertex ← Pick a random vertex from graph.Computation-Node()
10:    Find current machine that is running chosenVertex
11:    Deallocate chosenVertex from the machine
12:    newMachine ← Pick a random machine from list of Machines
13:    Assign newMachine to run chosenVertex
14: end function
```

Algorithm 3 Simulated Annealing Functions in the Routing Level

```
1: function initState(Graph, Machines)
2:     for Each pair of joined Computation-Node in the graph do
3:         Find which machines they are allocated on
4:         Take a random route from the pre-computed route between them
5:     end for
6: end function
7:
8: function genState(Graph, Machines)
9:     (Vertex1, Vertex2) ← Pick a random communicating Computation-Node in Graph
10:    Compute another route for Vertex1 and Vertex2 to communicate on
11: end function
```

is relatively minimal.

However, the above storage mechanism is not efficient since paths between two machines might overlap and thus, there will be a lot of duplication on the machines that are stored. Instead, we decide to store this information as a directed acyclic graph with two end-points; the two machines that we are interested on. In a cluster of 1000 nodes, we observe that the storage requirement reduces to around 6MB. Another point to note is that this Topology-Map is also aware of the bandwidth in the cluster. The pre-computation step takes around 3 minutes to compute. Topology-Map is stored as a hash-table with its key be $(\mathcal{M}_i, \mathcal{M}_j)$ where $\mathcal{M}_i, \mathcal{M}_j \in \mathcal{V}$ and its value be $\mathcal{K}_1, \mathcal{K}_2, \cdots \mathcal{K}_j$, where $\mathcal{K}_i$ is a sub-graph of $\mathcal{G}$, $\mathcal{G}'$. In each $\mathcal{G}'_i$, $\mathcal{M}_i$ and $\mathcal{M}_j$ are the only two end-points of the sub-graph.

At its core, our framework utilizes the Simulated Annealing [6] (SA) algorithm for approximating the global optimization problem on both the application and routing levels. SA is an iterative probabilistic technique for locating a good approximation of the best solution. The main idea of SA is to use heuristic which consider some neighbouring states $\mathcal{S}'$ of the current state $\mathcal{S}$, and probabilistically decides between moving the system to state $\mathcal{S}'$ or staying in state $\mathcal{S}$ on each iteration.

Since our framework works at two levels, a state different things at each level. In the routing level, a state $\mathcal{S}$ consists of $\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_j$ where $\mathcal{P}_i$ is a path in $\mathcal{G}$ where the end-points are a pair of communicating machines. Each $\mathcal{P}_i$ here is generated by traversing through one path in while neighbours of $\mathcal{S}$ is defined as generating exactly one different $\mathcal{P}_i$ from $\mathcal{G}'$.

In the placement level, a state $\mathcal{S}$ is a hash-table with the key as $\mathcal{M}_i$ and the value defined as the current placements of the worker tasks of the data processing framework while neighbours of $\mathcal{S}$ is defined as having exactly one worker task to a different machine.

The algorithm begins with a state initialization. When generating a new state in *genState*(), due to the enormous amount of possible neighbouring states that could exist in a datacenter with thousands of nodes, it is not feasible to randomly pick a neighbouring state as it may take a long time before converging. As such, we consider two heuristics when evaluating neighbouring states. From Chapter 1, recall that Computation-Node is defined as a vertex in the data-flow graph takes input data from the incoming edge and produces output data on the outgoing edge. Thus, our heuristic is as

follows. Firstly, we prefer COMPUTATION-NODE that have higher number of sink COMPUTATION-NODE to be allocated to machine that has more outgoing bandwidth. Secondly, we prefer COMPUTATION-NODE that have higher number of source COMPUTATION-NODE to be allocated to machine that has more incoming bandwidth. The rationale for this is that COMPUTATION-NODE with either high number of source COMPUTATION-NODEs or high number of sink COMPUTATION-NODEs will be a bottleneck for the job. In the routing level, there are also two heuristics that we are using. Firstly, we would prefer paths that have a lower number of hops. This will result in communication that will have lower latency and also probabilistically lower chance of disrupting existing and future jobs' traffic. Secondly, we would prefer paths that have the highest amount of available bandwidths so that the throughput of the jobs running in the datacenter are maximized.

Since a job is a graph of inter-connected tasks, observe that the network utilization of a final COMPUTATION-NODE equals to the most congested link of the upstream COMPUTATION-NODE in the graph. Since streaming frameworks allow multiple final COMPUTATION-NODE, the total throughput is equal to the sum of the throughput of each of the final COMPUTATION-NODE. All of these functionalities are performed in the *computeUtil* function.

Finally, we decide in the $doTransition()$ function whether to transition to the new state or stay in the old state. When the new total utilization, $newUtil$, is greater than the current total utilization, $currentUtil$, we set our current configuration, represented by $currentState$, to the new configuration, represented by $newState$. Otherwise, the transition is based on a probability function that logarithmically decrease as the number of iterations increase. For instance, assume that we arrive at a state $\mathcal{S}_{current}$ in which the throughput of the job is 10 MBps and the next state $\mathcal{S}_{next}$ that we plan to migrate to will result in the job having a throughput of 8 MBps. Although $\mathcal{S}_{next}$ has a lower throughput, we might probabilistically migrate there. However, the lower the throughput of $\mathcal{S}_next$ is, there is a lower probability that we will migrate there. Additionally, as time passes, if we encounter a similar situation where $\mathcal{S}_{next} \geq \mathcal{S}_{current}$, we assign lower probability for the migration.

The main flow of our framework is as follows. Firstly, a user will submit a job to the application (i.e. Storm, Hadoop, Spark, etc). Our scheduler that runs on the placement level will then consult the TOPOLOGY-MAP to evaluate which machines placement will yield the best bandwidth. Afterwards,

the application then request the SDN controller to setup the requested paths.

The SDN controller continuously runs in the background to monitor the network links. If there are any updates on the amount of bandwidth that a link can be supported or if there is a new link that is updated, the SDN controller will keep track of it and periodically send the updates to the application which then update its Topology-Map.

2.2 Failures

To support faster indexing whenever there is a link failure, the application also keeps track of a separate hash-table in which the key is an edge $\mathcal{E}_i$ in the graph while the value be $(\mathcal{M}_{1_1}, \mathcal{M}_{1_2}), (\mathcal{M}_{2_1}, \mathcal{M}_{2_2}), \cdots (\mathcal{M}_{j_1}, \mathcal{M}_{j_2})$ where $\mathcal{M}_{i_1}, \mathcal{M}_{i_2})$ is a pair of communicating machines in the topology who is using the link. Thus, when the application is notified that a link goes down, it will then update its Topology-Map and then generate another path.

Additionally, we take advantage of the failure detection mechanism that have usually been implemented by the application to detect failing machines. In this case, our scheduler will be aware of which machine(s) that fail and thus, which tasks that need to be rescheduled. We will then re-run our algorithm for these tasks. Additionally, until the failing machines will be brought back up again, it will be invalidated from the Topology-Map.

CHAPTER 3

IMPLEMENTATION

In this chapter, we show the implementation of our framework in two real-world system; Storm and Hadoop. We choose Storm because it is one of the most popular real-time frameworks that is being used by companies nowadays. We first give a background of Hadoop and Storm first before delving into the details of our implementation.

3.1 Storm

3.1.1 Design

Storm is a distributed real-time computation system that is able to reliably process streams. In here, we define a stream as an unbounded sequences of tuples that arrive in real-time at a node of the dataflow graph of Storm (e.g., Figure 1.1).

There are four main abstractions that exist in the Storm framework – bolts, spouts, topologies and stream groupings. A spout is a source of streams which can either generate tuples by itself or it can read from an external source. A bolt is an operator that takes input from any number of other spouts and/or bolts based on the dataflow graph and generates an output which can be send to any number of bolts. The function performed by bolt could range from filters, joins to aggregations. Thus, in the case of Figure 1.1, by the above definitions, V_1 and V_2 are spouts while V_3, V_4 and V_5 are the bolts.

Finally, a topology is a graph of spouts and bolts that are connected with stream groupings. Since Storm is a distributed framework, users can specify a particular bolt to have a number of tasks, each of which might be distributed to different physical machines. Stream groupings defines how a stream should be partitioned among these tasks. For instance, we could choose such that

the tuples are being randomly distributed among these tasks or we could group them by a particular field inside the tuples such that other tuples with same values in that field will be sent to the same machine.

When the Storm cluster first starts, each of the master node(s) runs a daemon called Nimbus that is responsible for code distribution and monitoring the status of the worker machines. Each of the worker machines runs a daemon called the Supervisor who will listen for works that are being assigned to them by the Nimbus. However, Nimbus and Supervisor machines do not communicate directly. They utilize Zookeeper [13], which is a centralized service that provides distributed synchronization and providing group services. This design allows the Nimbus and Supervisor machines to remain stateless and fail-fast.

3.1.2 Cross-Layer Scheduling Implementation

To incorporate our cross-layer scheduling, we modified Nimbus to contact the centralized controller to get information about the underlying topology that it is running on at start time. When a user submits a new job to Storm, it will place the spouts and the bolts according to first-level of the algorithm described in Algorithm 3. Then, it will request the centralized controller to allocate the paths for each communicating Computation-Node.

3.2 Hadoop

3.2.1 Design

Hadoop currently uses the YARN scheduler in which the main idea was to separate global resource management, Resource Manager (RM), from per-application resource management, Application Master (AM). The AM negotiates with the RM for each resource that it requires. Additionally, on each node in the cluster, there is a daemon called Node Manager (NM) that continually monitors the amount of resource left in the node and each AM will communicate with some of the NM to acquire some of the resources that the AM are entitled to by the RM.

We use Hadoop's MapReduce framework as a use-case for the design of our framework since MapReduce provides a divide-and-conquer data processing model, where large workloads are split into smaller tasks, each processed by a single server in a cluster (the map phase). The results of each task are sent over the cluster network (the shuffle phase) and merged to obtain the final result (the reduce phase). We focus the application of our algorithm on the shuffle phase as it is the phase that typically produce the most network footprint in a MapReduce job.

3.2.2 Cross-Layer Scheduling Implementation

Similar to Storm, to incorporate our cross-layer scheduling, we modified the RM to to contact the centralized controller to get information about the underlying topology that it is running on at start time. When a user submits a new job to Hadoop, RM will receive the number of mappers and reducers that the user wants. With this information, our cross-layer scheduling framework will place the mappers and reducers according to first-level of the algorithm described in Algorithm 3. Then, it will request the centralized controller to allocate the paths for each communicating mappers and reducers.

Since Hadoop uses the Map-Reduce paradigm, the problem statement boils down to deciding the placement of mappers and reducers instead of having to schedule in an arbitrary graph. When a new job arrives, it specifies the number of mappers, M, and the number of reducers, R and our algorithm will then be called each time a new job arrives. Since Hadoop keep tracks of the nodes that are currently free to schedule more jobs, $freeHost$, the first-level of SA will then iterate through $\binom{freeHost}{(M+R)}$ possible states. In each iteration, the first-level of SA will then call the second-level of SA. From our pre-computed routing paths, each mapper has k paths to communicate with each reducer. Thus, our second-level of SA will then iterate through k^2 possible states. In each iteration, the second-level of SA will compute the max-min fairness of the currently running jobs in conjunction with the potential paths and placements of the new job.

CHAPTER 4

EVALUATION

In this section, we evaluate our frameworks on top of two data center topologies, Fat-Tree [7] and Jellyfish [11]. The Fat-Tree topology consists of a collection of edge and aggregation switches that form a complete bipartite graph while Jellyfish is a random graph topology.

First, we start by evaluating the performance of Storm which has been incorporated with our algorithm in comparison with the vanilla Storm, followed by a similar performance evaluation in Hadoop in Emulab cluster. Due to scarcity of SDN testbeds, we emulated a centralized routing policies by implementing a software router in the Emulab cluster. Since we do not have access to hundreds of machines for experiments, we evaluated the scalability our cross-layer scheduling framework performance in a trace-driven simulation.

There are 3 important metrics for our algorithm performance; the throughput of each job that is running in the cluster, the average completion-time of each job which is the difference between the time it gets submitted to the time it gets to the time it finishes running and finally, the amount of time our algorithm takes for each scheduling decision.

4.1 Storm

Our Storm experiments are run on machines with a single 3 GHz processor, 2 GB of RAM and 200 GB for disks in the Emulab cluster running Ubuntu 12.04 while the link bandwidths are to 100 MBps.

Since Storm currently does not have any public traces, we generate synthetic topologies ourselves. Each of the topologies generated has two root nodes while each of the spouts and bolts in the topology have a random number of children which is selected by a Gaussian distribution with both

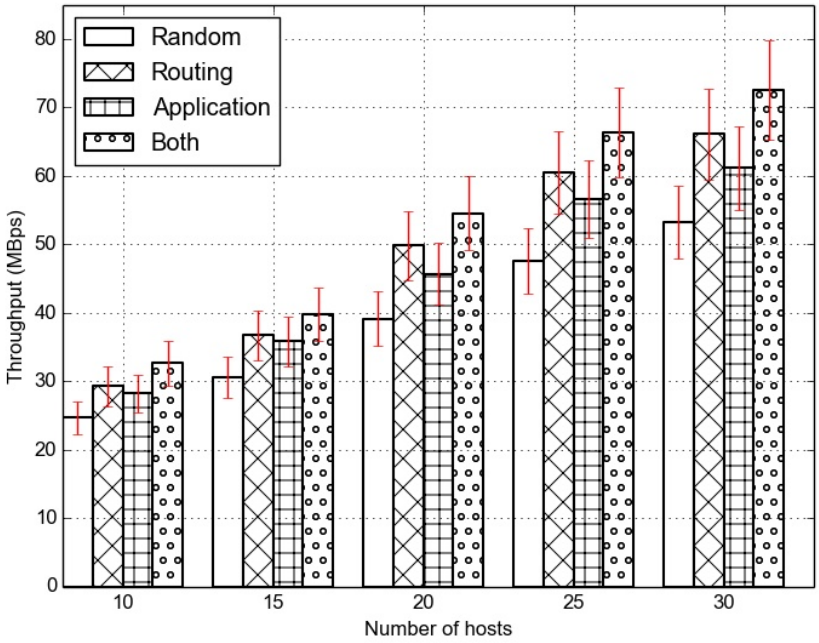

Figure 4.1: Throughput in Fat-Tree topology with varied routing strategies in Storm. Our cross-layer scheduling framework improves the throughput of vanilla Storm up to 35%.

μ and σ set to 2. The number of bolts in a topology is limited to 10. Additionally, each spout generates between 1 MB and 100 MB tuples of size 100 B records while the bolts only forward tuples. The latter choice allows us to focus on the impact of our frameworks on the network scheduling of applications rather than CPU or Memory.

Figure 4.1 and Figure 4.2 shows throughput of the jobs in Storm when run under the Fat-Tree and Jellyfish topology respectively where the label indicates which scheduling level is being used. "Random" indicates that it uses random placements and routing, Routing uses only routing-level scheduler, Application uses only application-level scheduler and Both uses both application and routing level scheduler. All of the improvements reported below are relative to the default scheduler's performance in Storm.

In the Fat-Tree topology (Figure 4.1), when the number of hosts is 10, we see that the average throughput of jobs running in the cluster is at 24 MBps when we do not run any of our scheduling. If we turn on the routing-level scheduler, the average throughput of the jobs then rises by 18.68% to 29 MBps and if we turn on the application-level scheduler, it rises by 14.4% to 28 MBps. When we have both of them turned on at the same time, the

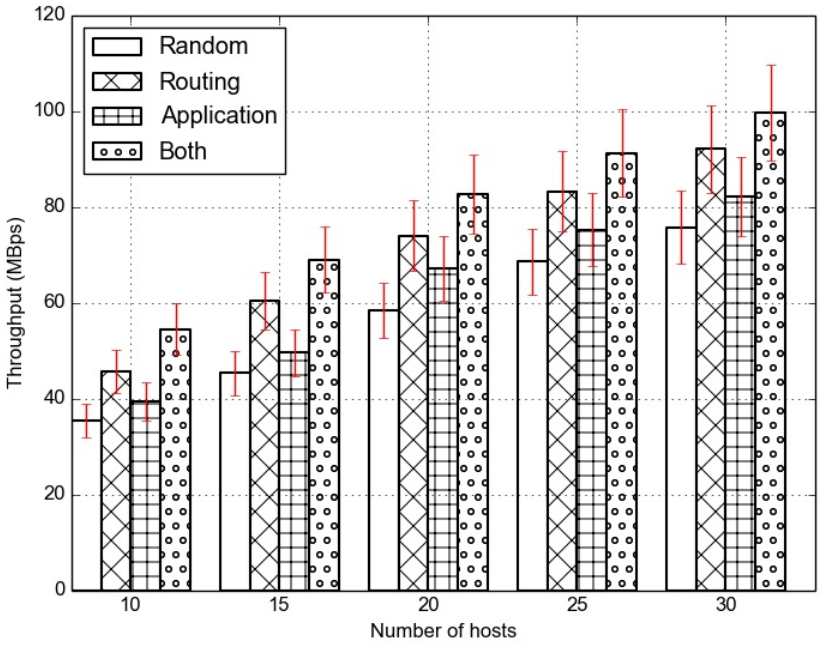

Figure 4.2: Throughput in Jellyfish topology with varied routing strategies in Storm. Our cross-layer scheduling framework improves the throughput of vanilla Storm up to 32%.

throughput further rises by 32.37%. With 30 hosts running in the cluster, throughput of the jobs rises by 35% from 53 MBps to 72 MBps when the two-level schedulers are used. As shown in the graph, the improvements in throughput increase as we scale the number of hosts. If we keep saturating the bandwidth of the network topology as we increase the number of hosts, the "Random" scheduler will sometimes schedule jobs at machines with low amount of outgoing bandwidth. Similar performance benefits are also seen in jobs running under the Jellyfish topology as we scale the number of hosts. With 30 hosts running in the cluster, throughput of the jobs rises by 20% and 9% as we turn on the routing-level and application-level scheduler. When we have scheduler at both levels running, the throughput of the jobs rises by 32% cumulatively as compared to the default scheduler that is running in Storm.

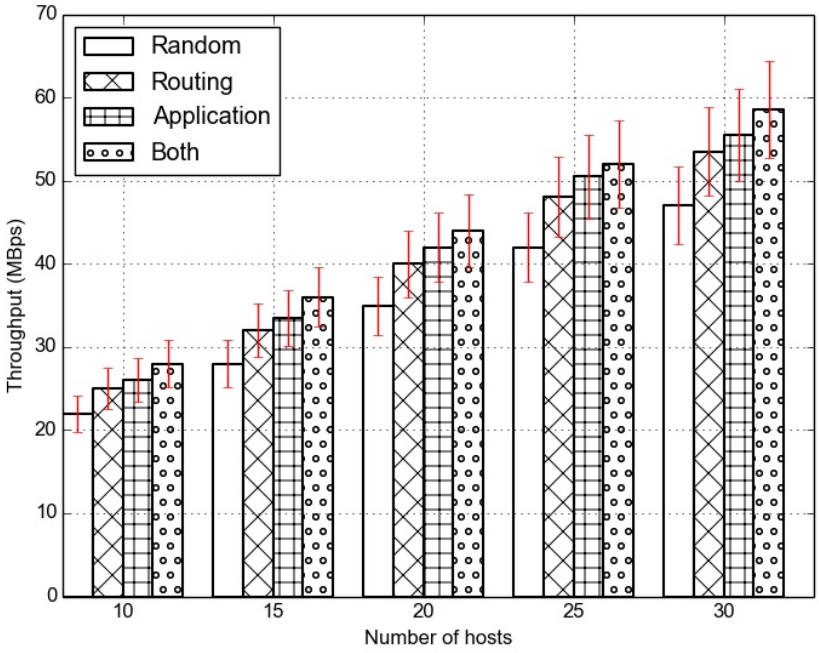

Figure 4.3: Throughput in Fat-Tree topology with varied routing strategies in Hadoop. Our cross-layer scheduling framework improves the throughput of vanilla Hadoop by up to 23%.

4.2 Hadoop

In this section, we demonstrate how our framework works with Hadoop paradigm. Similar to the Storm experiments, our Hadoop experiments are run on machines with a single 3 GHz processor, 2 GB of RAM and 200 GB disk in the Emulab cluster running Ubuntu 12.04 with the link bandwidth set to 100 MBps. The workload used for the following experiments is the Facebook workload provided by by the SWIM benchmark [14]. The data-size generated by the mappers is between 1 MB to 100 MB. Similar to the Storm experiments, the jobs that we run only forward data from mappers to reducers.

Figure 4.3 and Figure 4.4 shows throughput of the jobs in Hadoop when run under the Fat-Tree and Jellyfish topology respectively where the label indicates which scheduling level is being used. The label Random indicates that it uses random placements and routing, while "Routing" uses only routing-level scheduler, "Application" uses only application-level scheduler and "Both" uses both application and routing level scheduler. All of the improvements reported below are relative to the default scheduler's perfor-

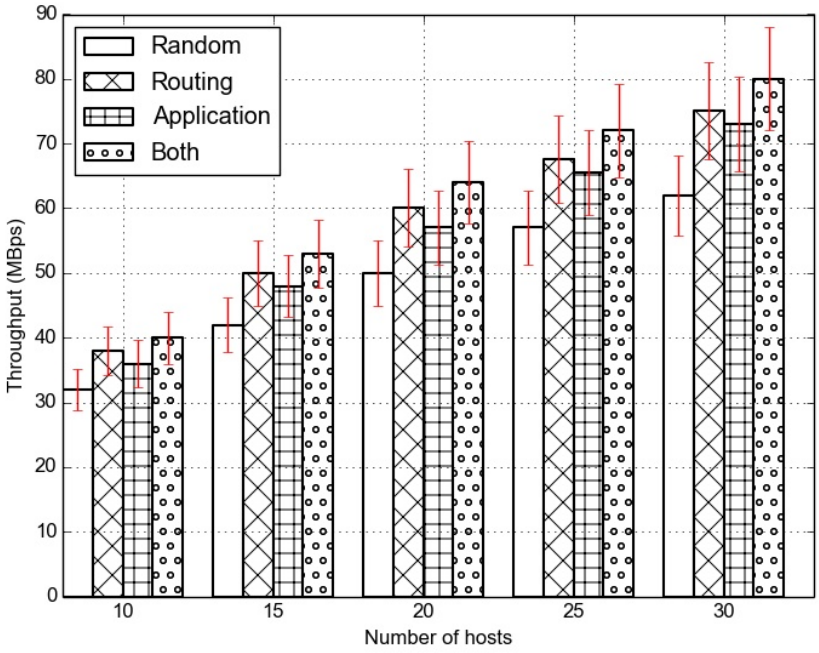

Figure 4.4: Throughput in Jellyfish topology with varied routing strategies in Hadoop. Our cross-layer scheduling framework improves the throughput of vanilla Storm by up to 26%.

mance in Hadoop.

In the Fat-Tree topology (Figure 4.3), when the number of hosts is 10, we see that the average throughput of jobs running in the cluster is at 22 MBps when we do not run any of our scheduling. If we turn rises by 18.18% to 26 MBps. When we have both of them turned on at the same time, the throughput further rises by 22.31%. As shown in the graph, the improvements in throughput can still be observed when we scale the number of hosts. With 30 hosts running in the cluster, throughput of the jobs rises by 23.53% from 47 MBps to 59 MBps when the two-level schedulers are used. Similar performance benefits are also seen in jobs running under the Jellyfish topology (Figure 4.4) as we scale the number of hosts. With 30 hosts running in the cluster, throughput of the jobs rises by 20.97% and 17.74% as we turn on the routing-level and application-level scheduler. When we have scheduler at both levels running, the throughput of the jobs rises by 26.03% cumulatively as compared to the default scheduler that is running in Storm.

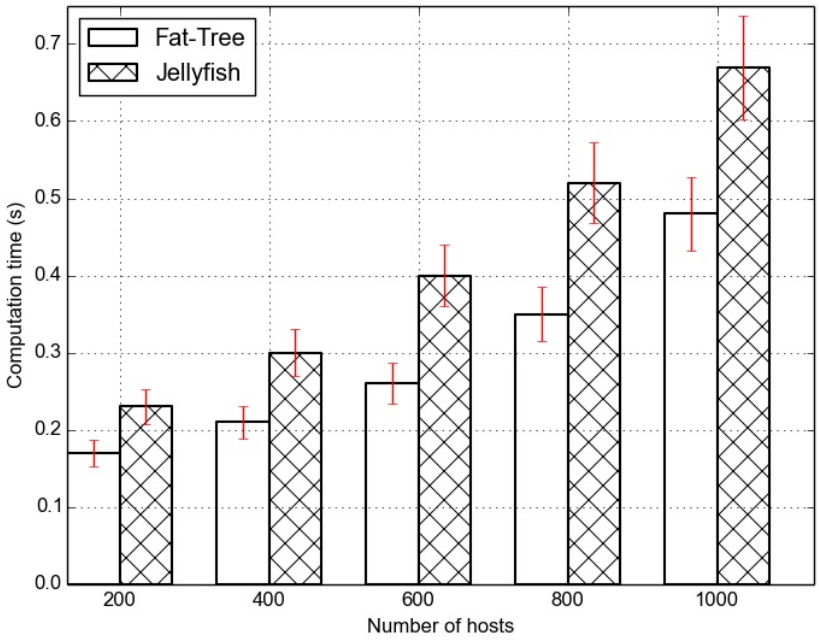

Figure 4.5: Scheduling time for cross-layer scheduling in cloud computing systems with Map-Reduce data flow. At 1000 nodes, our cross-layer scheduling framework takes 0.48 s and 0.67 s per scheduling decision in Fat-Tree and Jellyfish respectively.

4.3 Simulation

For our trace-driven simulation, all experiments are run on a single machine with 4-core 2.50 GHz processor and the bandwidth of each link in the simulation is set to 100 MBps.

Figure 4.7 and Figure 4.8 plots the computation time for our algorithm as we scale the number of hosts in the cluster from 200 to 1000. As mentioned in Section 2, our algorithm runs every time a new job comes in at the rate of . Our plot shows the average time across all the calls to our algorithm.

Under stream processing data flow (Figure 1.1), each scheduling decision on average only takes 0.28 s in a 200-node Jellyfish topology. If we scale it up to 1000 nodes, each scheduling decision takes 0.74 s. Similarly, in the Fat-Tree topology, each scheduling decision takes 0.20 s in average to perform and if we scale it up to 1000 nodes, it takes 0.53 s in average.

Under Map-Reduce based frameworks, each scheduling decision in average only takes 0.23 s in a 200 nodes Jellyfish topology. If we scale it up to 1000 nodes, each scheduling decision takes 0.67 s. Similarly, in the Fat-Tree

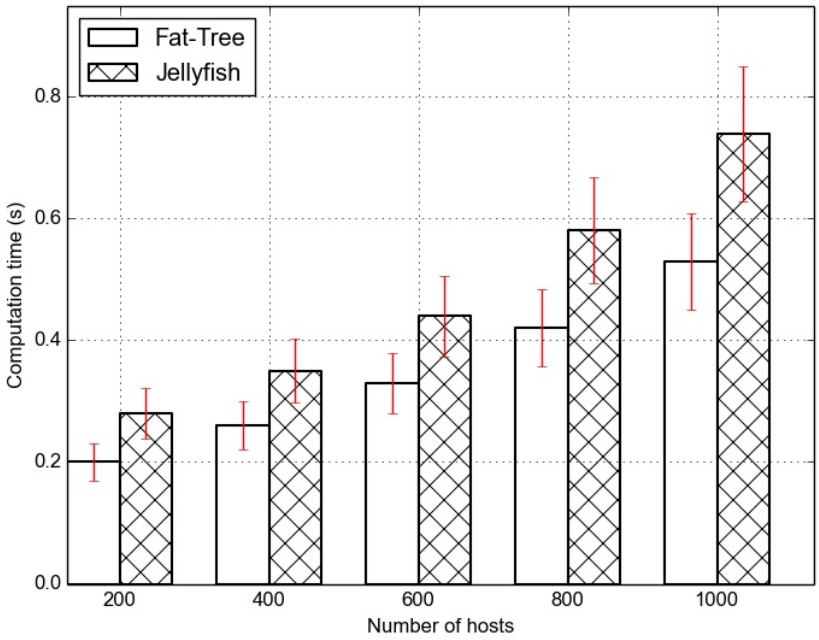

Figure 4.6: Scheduling time for cross-layer scheduling in cloud computing systems with stream processing data flow. At 1000 nodes, it only takes 0.53 s and 0.74 s per scheduling decision in Fat-Tree and Jellyfish respectively.

topology, each scheduling decision takes 0.17 s in average to perform and if we scale it up to 1000 nodes, it takes 0.48 s in average.

Both of the above graphs show that our algorithm is scalable with a large number of hosts in the cluster, by still being able to make sub-second scheduling decision under 1000 nodes cluster.

Finally, Figure 4.7 and Figure 4.8 show a CDF plot of job completion time improvements in Hadoop data flow and Storm data flow respectively in comparison with a scheduler that picks random machines while picking the shortest routing path. Each of the plots also shows the Fat-Tree and Jellyfish topology that we are running on. Figure 4.7 shows that our cross-layer scheduling framework improves Hadoop's job completion time by 34% (and 40%) at 50^{th} (and 75^{th}) percentile while Figure 4.8 shows that our cross-layer scheduling framework improves Storm's job completion time by 38% (and 42%) at 50^{th} (and 75^{th}). This shows that our algorithm improves

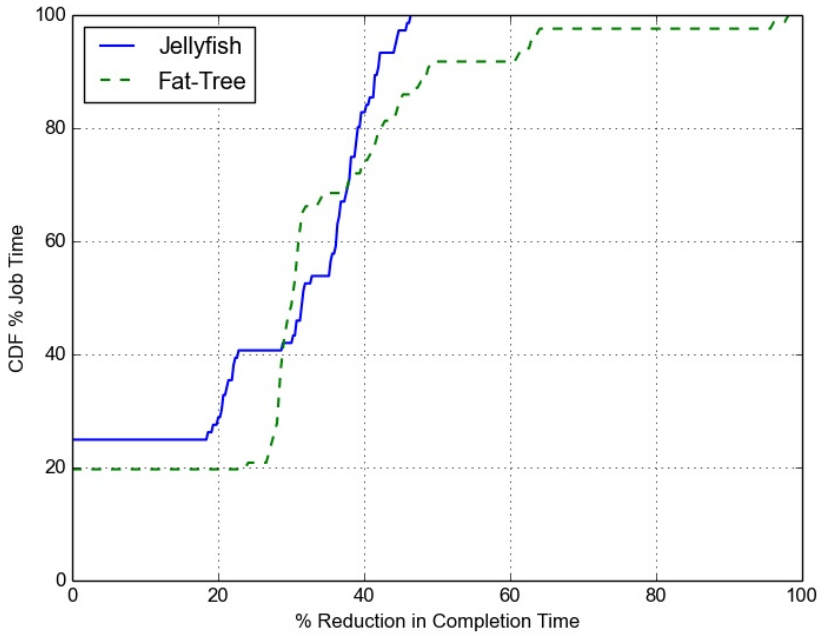

Figure 4.7: CDF of job completion time improvements in Hadoop at 1000 nodes when cross-layer scheduling framework is activated.

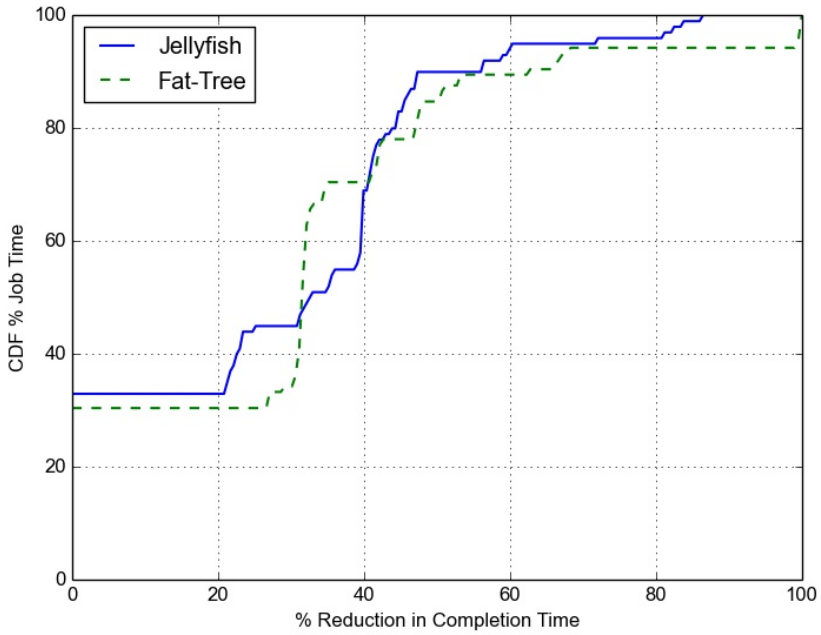

Figure 4.8: CDF of job completion time improvements in Storm at 1000 nodes when cross-layer scheduling framework is activated.

CHAPTER 5

RELATED WORK

Many of the previous works looked into improving the throughput of batch processing frameworks such as Hadoop [1] that utilizes the Map-Reduce paradigm [3]. In the area of Hadoop scheduling, there has been two main directions of research that have been taken; *computation scheduling* and *network scheduling*.

Computation Scheduling: Several works look to improve job scheduling by preserving data locality [15, 16, 17, 18], propose scheme to maintain fairness across CPU and memory resources [19, 20]. Other works have tried to pack tasks requiring different resources into different machines [21]. Jockey [22] takes it one step further to precomputes statistics using a simulator that captures the jobs complex internal dependencies and dynamically adjusts resource allocation in the shared cluster. Techniques proposed by these papers however are and thus orthogonal to our proposal since they are targeted at managing resources such as CPU and memory which, unlike network, is local and not shared.

Application-Level Scheduling: There are some that have looked into network scheduling in Hadoop. One of the strategies proposed by Mantri [23] include network-aware placement of tasks in which each job manager in Hadoop places tasks so as to minimize the load on the network and avoid self-interference among the tasks. Mantri also takes into account the cost of moving data so that no task is being started at location that has little bandwidth. More importantly, the scheduling policies proposed by these papers are restricted to Hadoop style of communication whereas in streaming frameworks, the dataflow is much more expressive. There have been many streaming frameworks that have been built such as Spark [24], Naiad [25], TimeStream [26] but none of the schedulers that they use involve exposing

information about the underlying network topology.

Routing-Level Scheduling : There has been a lot of previous works such as Orchestra [27] and Seawall [28] which propose to improve the shuffle phase by scheduling flows using a weighted fair scheme. Oktopus [29] and Second-Net [30] propose static reservations throughout the network to implement bandwidth guarantees for the hose model and pipe model, respectively. e main drawback of reservation systems is that they do not achieve the work conservation property, since the unused bandwidth is not shared between tenants. Gatekeeper [31] proposes a per-VM hose model with work conrvation. Gatekeeper uses a hypervisor-based mechanism, which, however, works only for full bisection-bandwidth networks. FairCloud [32] improves upon previous approaches by introducing multiple policies to either achieve link-proportionality or congestion-proportionality. Many of these works can be leveraged to work in our network-level scheduler.

Additionally, many of these previous works could not be applied directly to the streaming frameworks in which the data flow representation is in the form of graph and thus, much more flexible than the Map-Reduce paradigm that we have seen so far.

Software-Defined Networking : Before SDN, a number of methods for optimizing the network to improve application performance or availability have been considered previously. Examples of these include providing custom instances of routing protocols to applications [33], or even allowing applications to embed code in network devices to perform application processing [34]. However, these works do not allow the fine granularity of network monitoring like SDN does. There has been many works that strives to improve the area of SDN but they are either targeted on either improving the scalability of SDN (e.g., enabling a hierarchy of policies to SDN controller [35]), enforcing correctness in SDN (e.g., enabling consistent updates in SDN [35]) or enable SDN to work with broader range of the current network protocols (e.g., radio network [36], cellular network [37]). [38] has also considered a two-level optimization for data processing frameworks. We incorporate their proposals batch processing of updates, i.e. a group of tasks submitted in a period T will be processed together to minimize routes reconfiguration. However, they have not shown any concrete algorithms and experimental results. Flow-

comb [39] has shown that Hadoop could improve its performance by having application-level hints being sent to an SDN centralized controller although the paper only explores routing level optimization, and not placement level.

VM Placements: Many of the works in the VM placements have tried tackling similar problem. [40, 41] proposes a linear programming optimization that tries to migrate. They leverage Markov approximation that attempts to minimize the amount of VM migrations. [42] proposes a shadow routing based VM placement algorithm. Similar to [40], [42] uses Markov Chains for combinatorial optimization. However, many of the techniques used in these works involve algorithms that utilize linear programming optimization which consume a lot of time and even their online algorithm proposals are still in the order of seconds which are not sufficient for our requirement of satisfying interactive applications.

Cross-Layer Scheduling: Cross-layer scheduling is not a new approach. It has been successfully tested in other systems in computer science. Many of today's Operating Systems use two-level scheduling to separates decisions on global resource allocation from application-specific resource [43, 44, 45]. The high performance computing (HPC) community has been also been using the technique to manage its clusters [46, 47]. Virtual machine clouds such as Amazon EC2 [48] and Eucalyptus [49] also use the technique to isolate applications while providing a low-level abstraction (VMs). Mesos [50] takes it further and allows frameworks running on top of the cluster to be highly selective about the task placement. However, none of them has tried to bridge the information that is available in the distributed systems level with that in the networking level which is the problem that we are tackling in this thesis.

CHAPTER 6

CONCLUSION

In this thesis, we identified three challenges that need to be overcome to be able to run generic distributed computing applications efficiently. Firstly, applications need to be aware of the underlying networking topology so that it is able to make better scheduling decisions with respect to the amount of bandwidth that is available between machines. Secondly, with datacenters today comprising of hundreds and even thousands of machines, finding the most optimal machines that a user's job demands is a problem. This problem is further exacerbated by users' demand for more interactive jobs which implies that scheduling time should be in sub-seconds range. Finally, the scheduler needs to be able to support both batch-processing frameworks with Map-Reduce paradigm or stream processing frameworks which have an arbitrary directed graph.

Our proposed cross-layer throughput optimization framework addresses the above challenges. We separate the scheduling logic in two levels: application and routing. Our routing-level scheduler is realized through SDN which allows us to install arbitrary forwarding rules in routers. In each of the level, we leverage one of the most well-known approximation technique in a state-space search problem, Simulated Annealing. Combined with the heuristics that we devised to explore the possible states more intelligently, we have shown that our algorithm completes in hundreds of milliseconds while only requiring around 6MB of storage. This addresses the second challenge. Finally, since we use Simulated Annealing, which is a generic probabilistic state-space search, we made no assumption about the underlying data-path, thus automatically solving the third problem. As shown in Figure 4.4 and Figure 4.2, our experimental results have shown that we are able to improve performance of streaming applications in Storm by 32% and batch processing application in Hadoop by 29%.

6.1 Future Work

Since workers in streaming frameworks are mostly stateless, migration of workers could be executed much faster than for batch processing frameworks. Thus, an area for future work will be to evaluate whether we need to migrate some tasks when there is a change in the streaming workload or if there is a new job that needs to be executed. If we do decide that we should migrate, the next question that we need to answer is where should we migrate these tasks to.

Finally, Sparrow [51] has demonstrated that it is possible to perform decentralized scheduling in 1ms with a performance within 12% of an ideal scheduler by using random sampling. Although the resources that Sparrow schedules are limited to CPU and memory, we should explore ways that we could decentralize our scheduling logic of the network resources by intelligently partitioning the data flow graph.

REFERENCES

[1] "Apache Hadoop," Oct. 2013. [Online]. Available: http://www.hadoop.apache.org/

[2] "Storm, distributed and fault-tolerant realtime computation." [Online]. Available: http://storm-project.net/

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[5] M. Appelman and M. de Boer, "Performance analysis of openflow hardware," 2012.

[6] P. J. Van Laarhoven and E. H. Aarts, *Simulated annealing*. Springer, 1987.

[7] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. 34, no. 10, pp. 892–901, Oct. 1985.

[8] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[9] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 75–86.

[10] L. Gyarmati and T. A. Trinh, "Scafida: a scale-free network inspired data center architecture," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 4–12, 2010.

[11] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: networking data centers randomly," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 17–17.

[12] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, vol. 8, 2010, pp. 11–11.

[14] "Statistical workload injector for mapreduce (SWIM)," Aug. 2013. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki

[15] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*. ACM, 2010, pp. 265–278.

[16] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. ACM, 2009, pp. 261–276.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[18] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proceedings of the 6th Conference on Computer Systems*. ACM, 2011, pp. 287–300.

[19] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: fair allocation of multiple resource types," in *Proceedings of the 11th USENIX Symposium on Network Systems Design and Implementation*, 2011.

[20] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework," in *Proceedings of the 31st IEEE International Conference on Computer Communications*. IEEE, 2012, pp. 1206–1214.

[21] T. Sandholm and K. Lai, "Mapreduce optimization using regulated dynamic prioritization," in *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*. ACM, 2009, pp. 299–310.

[22] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 2012, pp. 99–112.

[23] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using Mantri." in *Operating Systems Design and Implementation*, vol. 10, no. 1, 2010, p. 24.

[24] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737 pp. 423–438.

[25] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 439–455.

[26] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 1–14.

[27] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," *ACM SIGCOMM-Computer Communication Review*, vol. 41, no. 4, p. 98, 2011.

[28] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Seawall: performance isolation for cloud datacenter networks," in *Proceedings of the 2nd USENIX Conference on Hot topics in Cloud Computing*. USENIX Association, 2010, pp. 1–1.

[29] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 242–253.

[30] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Proceedings of the 6th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 2010, p. 15.

[31] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes, "Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks," *Proceedings of the 3rd USENIX Workshop on I/O Virtualization*, 2011.

[32] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: sharing the network in cloud computing," in *Proceedings of the ACM SIGCOMM 2012 conference*. ACM, 2012, pp. 187–198.

[33] P. Chandra, A. Fisher, C. Kosak, T. E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, "Darwin: Customizable resource management for value-added network services," in *Proceedings of the 6th International Conference on Network Protocols*. IEEE, 1998, pp. 177–188.

[34] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A survey of active network research," *Communications Magazine, IEEE*, vol. 35, no. 1, pp. 80–86, 1997.

[35] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 2011, p. 7.

[36] A. Gudipati, D. Perry, L. E. Li, and S. Katti, "Softran: Software defined radio access network," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. ACM, 2013, pp. 25–30.

[37] L. E. Li, Z. M. Mao, and J. Rexford, "Toward software-defined cellular networks," in *Proceedings of the European Workshop on Software Defined Networking*. IEEE, 2012, pp. 7–12.

[38] G. Wang, T. Ng, and A. Shaikh, "Programming your network at runtime for big data applications," in *Proceedings of the 1st Workshop on Hot Topics in Software Defined Networks*. ACM, 2012, pp. 103–108.

[39] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu, "Transparent and flexible network management for big data processing in the cloud," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX.

[40] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang, "Joint VM placement and routing for data center traffic engineering," in *Proceedings of the 31st IEEE International Conference on Computer Communications*. IEEE, 2012, pp. 2876–2880.

[41] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proceedings of the 29th IEEE International Conference on Computer Communications*. IEEE, 2010, pp. 1–9.

[42] Y. Guo, A. L. Stolyar, and A. Walid, "Shadow-routing based dynamic algorithms for virtual machine placement in a network cloud," in *Proceedings of the 32nd IEEE International Conference on Computer Communications*. IEEE, 2013, pp. 620–628.

[43] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*. Wiley, 2013, vol. 8.

[44] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, "Tessellation: Space-time partitioning in a manycore client OS," *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, vol. 3, p. 2009, 2009.

[45] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Operating Systems Design and Implementation*, vol. 96, no. 22, 1996, pp. 107–121.

[46] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. ACM, 2006, p. 8.

[47] S. Zhou, "Lsf: Load sharing in large heterogeneous distributed systems," in *I Workshop on Cluster Computing*, 1992.

[48] "Amazon EC2." [Online]. Available: http://aws.amazon.com/ec2

[49] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus open-source cloud-computing system," in *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, 2009, pp. 124–131.

[50] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, 2011, pp. 22–22.

[51] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 69–84.